Build an OpenAI and Milvus Powered RAG System for Healthcare Domain

Test Project for Superteams.ai

Purpose: To guide users through the process of building a Retrieval-Augmented Generation (RAG) system for the healthcare domain using OpenAI and Milvus.

Technology Stack

- **Programming Languages**: Python
- **Frameworks & Libraries**:
    - **OpenAI GPT-4**: For generating responses and natural language understanding.
    - **Milvus**: For vector similarity search and managing the embedding database.
    - **PyMilvus**: Python SDK for Milvus.
    - **Hugging Face Transformers**: For various NLP tasks and model embeddings.
    - **Flask/FastAPI**: For building RESTful APIs.
    - **Docker**: For containerization and easy deployment.
    - **Streamlit or Gradio**: For UI creation.
- **Databases**:
    - **Milvus**: Vector database.

Project Goals

1. **Build a Scalable RAG System**: Leverage OpenAI and Milvus to create a system that can retrieve and generate accurate and relevant healthcare information.
2. **Integration with Vector Database**: Use Milvus to store and retrieve embeddings efficiently.
3. **Develop an Intuitive UI Using Streamlit or Gradio**: Provide a user-friendly interface for querying the system.

Possible Dataset

- **Sources**:
    - **MIMIC-III**: Clinical database of de-identified health-related data.
    - **Clinical Trials**: Database of clinical studies conducted around the world.
- **Data Preparation**:
    - **Text Preprocessing**: Tokenization, normalization, and cleaning of text data.
    - **Embedding Generation**: Use pre-trained NLP models to generate embeddings for the text data.

Outline for the Tutorial

1. **Introduction**
    - Overview of RAG systems
    - Importance of RAG in the healthcare domain
    - Objectives of this tutorial
2. **Technology Stack Overview**
    - Detailed explanation of the chosen technology stack
    - Installation and setup instructions for each component
3. **Dataset Preparation**
    - Selecting and sourcing healthcare datasets
    - Data preprocessing techniques
    - Generating embeddings using Hugging Face Transformers
4. **Setting Up Milvus**
    - Installing Milvus and PyMilvus
    - Configuring Milvus for vector storage
    - Indexing and inserting embeddings into Milvus
5. **Integrating OpenAI GPT-4**
    - Accessing OpenAI API
    - Integrating GPT-4 with the system for generating responses
    - Prompt engineering and optimizing for healthcare queries
6. **Building the RAG System**
    - Overview of the RAG architecture
    - Implementing the retrieval component using Milvus
    - Integrating the generation component using OpenAI GPT-4
    - Combining retrieval and generation to form the RAG pipeline
7. **Developing the API**
    - Setting up Flask/FastAPI for the RAG system
    - Creating endpoints for querying and managing the system
    - Handling requests and responses efficiently
8. **Results and Validation**
    - Showcase results on a UI
    - Validate the accuracy and relevance of responses
9. **Deployment and Scalability**
    - Containerizing the application using Docker
10. **Conclusion**
    - Recap of what was built
    - Future enhancements and improvements
    - Final thoughts and next steps

Introduction

Overview of RAG Systems

Retrieval-Augmented Generation (RAG) systems enhance large language models (LLMs) by integrating external, verifiable knowledge sources. This is crucial for domains like healthcare where accuracy and consistency are paramount.

Challenges in Machine-Generated Content:

- **Hallucination**: Models may produce repetitive or nonsensical content due to limited training data.
- **Inconsistent Data**: Models can generate different answers to the same question.
- **Repetitive Data**: Repeated or similar responses can be problematic in sensitive domains.

RAG Solution: By integrating external knowledge with LLMs, RAG systems ensure more accurate and reliable outputs. In this project, RAG enhances vector database search in the medical domain to improve knowledge accuracy.

Technologies incorporated

Application Interfaces

1. Web frameworks
    - **Gradio**
        - **Purpose**: Provides a quick and interactive web interface for users to interact with the machine learning model.
        - **Features**: Text input fields, model integration, public link deployment.
        - **Use Case**: Simplifies deploying and sharing a demo for model inference.
    - **Streamlit**
        - **Purpose**: Creates a web application for complex or customizable user interactions with the model.
        - **Features**: Supports text input, buttons, displays, integration with external services.
        - **Use Case**: Ideal for building a feature-rich application for dynamic user interactions.
2. **Machine Learning and Embeddings**
    - **OpenAI GPT-4**
        - **Purpose**: Provides advanced natural language processing capabilities.
        - **Features**: Text generation, API integration.
        - **Use Case**: Enhances responses by generating detailed and contextually accurate information.
    - **Transformers (Hugging Face)**

        - **Purpose**: Offers pre-trained models and tokenizers for generating embeddings.
        - **Features**: BERT-based model for embeddings, tokenizer for text processing.
        - **Use Case**: Creates embeddings for similarity search and contextual understanding.
3. **Data Storage and Retrieval**
    - **Dataset**: The MIMIC-III dataset, a publicly available database of de-identified health data, including clinical notes, laboratory results, and demographics.
    - **Source**: Available from Kaggle or directly from the MIMIC-III website.
    - **Preprocessing Steps**:
        - **Loading Data**: Load data into a Pandas DataFrame.
        - **Data Cleaning**: Handle missing values, remove duplicates, address inconsistencies.
        - **Feature Extraction**: Extract relevant columns.
        - **Text Normalization**: Clean and preprocess text data.

Example Code for Preprocessing:
python

```
import pandas as pd

def preprocess_mimic_data(file_path):
    """Load and preprocess MIMIC-III data."""
    mimic_data = pd.read_csv(file_path)
    mimic_data = mimic_data[['text_column', 'response_column']]  # Adjust according to actual column names
    return mimic_data
```

4.
5. **Data Vectorization and Storage**
    - **Data Vectorization**:
        - **Purpose**: Convert text data into numerical vectors using embeddings.
        - **Implementation**: Generate embeddings using a pre-trained Hugging Face Transformer model (e.g., BERT).
    - **Storage**:
        - **Milvus**: High-performance vector database for managing embeddings.
        - **Integration**: Store and retrieve embeddings efficiently.

Example Code for Inserting Embeddings into Milvus:
python

```
from pymilvus import connections, Collection, FieldSchema, CollectionSchema, DataType

# Connect to Milvus
connections.connect("default", host="localhost", port="19530")

# Define a Milvus collection
```

```
fields = [
    FieldSchema(name="id", dtype=DataType.INT64, is_primary=True, auto_id=True),
    FieldSchema(name="embedding", dtype=DataType.FLOAT_VECTOR, dim=768)  # Adjust
dimension as needed
]
schema = CollectionSchema(fields)
collection = Collection(name="healthcare_data", schema=schema)

# Function to insert embeddings into Milvus
def insert_embeddings(texts):
    """Insert embeddings into Milvus."""
    embeddings = [generate_embedding(text).tolist() for text in texts]
    collection.insert([embeddings])
    index_params = {
        "index_type": "IVF_FLAT",
        "metric_type": "L2",
        "params": {"nlist": 100}
    }
    collection.create_index("embedding", index_params)
```

6. **Tunneling and Access**
    - **LocalTunnel**
        - **Purpose**: Provides a public URL to access local applications.
        - **Features**: Exposes local servers to the internet, easy setup.
        - **Use Case**: Allows remote access to the Streamlit app running locally.
7. **Development and Deployment**
    - **Python**
        - **Purpose**: Primary language for scripting and building the application.
        - **Features**: Libraries for ML models, web development, API interactions.
        - **Use Case**: Core language for implementing the application logic.
    - **Google Colab**
        - **Purpose**: Cloud-based notebook environment for Python code development.
        - **Features**: Free access, GPU support, easy sharing.
        - **Use Case**: Ideal for prototyping and testing the application.

Workflow Summary

1. **Data Preparation and Storage**:
    - Insert customer support queries into Milvus after generating embeddings using Hugging Face Transformers.
2. **Interface Development**:

    - **Gradio**: Set up a simple interface to quickly test and share customer support query responses.
    - **Streamlit**: Develop a more interactive web app for dynamic query handling and display.
3. **Deployment and Access**:
    - Use **LocalTunnel** to expose the Streamlit app to the internet for remote access.
    - **Google Colab** serves as the development environment for running and testing the entire stack.
4. **Integration**:
    - **Gradio** and **Streamlit** connect to the same backend logic for processing queries.
    - **OpenAI GPT-4** and **Transformers** provide functionality for generating and embedding text.
    - **Milvus** handles vector storage and retrieval to support querying capabilities.

Step-by-Step Implementation

Step 1: Install Docker

- Ensure Docker is installed on your local machine. Download and install Docker.

Step 2: Create Docker Compose File

- Create a `docker-compose.yml` file to define the Milvus service.

docker-compose.yml:

```
version: '2.0'
services:
  milvus:
    image: milvusdb/milvus:2.2.2
    container_name: milvus
    ports:
      - "19530:19530"
    environment:
      - "MILVUS_DB_ADDRESS=localhost:19530"
      - "MILVUS_DB_PORT=19530"
    volumes:
      - milvus_data:/var/lib/milvus

volumes:
  milvus_data:
```

Step 3: Start Milvus

Run the command:
bash

```
docker-compose up -d
```

Verify Milvus is running:
bash

```
docker ps
```

Step 4: Install and Start ngrok

Download ngrok: ngrok's official website.

Start ngrok:
bash

```
./ngrok tcp 19530
```

Get the ngrok URL: Note the hostname and port provided by ngrok.

Step 5: Connect to Milvus from Google Colab

Install packages:
bash

```
!pip install openai pymilvus==2.2.2 transformers flask fastapi uvicorn gradio grpcio==1.63.0
```

Import libraries and configure connection:

```
import openai
from pymilvus import connections, Collection, FieldSchema, CollectionSchema, DataType
from transformers import AutoTokenizer, AutoModel
import torch
import gradio as gr

# Connect to Milvus
ngrok_host = '0.tcp.ngrok.io'
ngrok_port = '12345'
connections.connect("default", host=ngrok_host, port=ngrok_port)

# Define schema for Milvus collection
```

```
fields = [
    FieldSchema(name="id", dtype=DataType.INT64, is_primary=True, auto_id=True),
    FieldSchema(name="embedding", dtype=DataType.FLOAT_VECTOR, dim=768)
]
schema = CollectionSchema(fields)
collection = Collection(name="healthcare_data", schema=schema)

# Initialize OpenAI API key
openai.api_key = "your-api-key"

# Initialize tokenizer and model
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")

# Function to generate embeddings
def generate_embedding(text):
    inputs = tokenizer(text, return_tensors="pt")
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state.mean(dim=1).detach().numpy()
    return embeddings

# Function to insert embeddings into Milvus
def insert_embeddings(texts):
    embeddings = [generate_embedding(text).tolist() for text in texts]
    collection.insert([embeddings])
    index_params = {
        "index_type": "IVF_FLAT",
        "metric_type": "L2",
        "params": {"nlist": 100}
    }
    collection.create_index("embedding", index_params)

# Function to retrieve similar documents
def retrieve_similar_documents(query_embedding, top_k=5):
    search_params = {"metric_type": "L2", "params": {"nprobe": 10}}
    results = collection.search([query_embedding], "embedding", search_params,
top_k=top_k)
    return results

# Function to generate a response using OpenAI GPT-4
def generate_response(prompt):
    response = openai.Completion.create(
        engine="gpt-4",
        prompt=prompt,
        max_tokens=150
    )
    return response.choices[0].text.strip()
```

```
# Function to generate an augmented response
def generate_augmented_response(query):
    query_embedding = generate_embedding(query).tolist()
    similar_docs = retrieve_similar_documents(query_embedding)
    context = " ".join([str(doc) for doc in similar_docs])
    augmented_prompt = f"Given the following medical context, provide a detailed response:
{context}"
    return generate_response(augmented_prompt)
```

Step 6: Setting Up Streamlit

Example Data:

```
example_texts = [
    "Diabetes symptoms can vary depending on the type of diabetes.",
    "Metformin, commonly used to treat type 2 diabetes, can have side effects.",
    "Several clinical trials are currently being conducted to find new treatments for Alzheimer's
disease."
]

# Insert example embeddings into Milvus
insert_embeddings(example_texts)
```

Run Gradio App:

```
def run_gradio_app():
    iface = gr.Interface(fn=generate_augmented_response, inputs="text", outputs="text")
    iface.launch()

run_gradio_app()
```

Summary

1. **Install Docker** and create a docker-compose.yml file to run Milvus locally.
2. **Start Milvus** using Docker Compose.
3. **Use ngrok** to expose the Milvus port to the internet.
4. **Run the complete code** in Google Colab, replacing placeholders like ngrok_host, ngrok_port, and openai.api_key with actual values.

This setup will configure Milvus, connect it to your Colab notebook, and run the Gradio interface for querying the system.

In the following diagram, the system is summarized to give an idea of the project tutorial.